Verifying GraalVM Optimizations — Why?

Brae Webb

October 2021

Abstract

At the University of Queensland, we're working to verify the optimization passes of the GraalVM compiler. This article conveys the advantages of verifying a polyglot partially evaluating compiler, such as GraalVM. Specifically, we explore how optimizations enable polyglot behaviour, and how optimizations become shared among host languages. This is demonstrated through the manual partial evaluation of a hypothetical Simple Arithmetic Language (SAL).

1 GraalVM and Truffle

The GraalVM compiler is a modern, state-of-the-art compiler for JVM bytecode. At face value, the major advantage of verifying GraalVM optimizations is that it hosts an extensive suite of optimizations. However, when investigating Truffle, the compilers approach to implementing new languages, the advantages becomes apparent.

As a polyglot compiler, GraalVM is able to act as a compiler for a wide range of languages, all the way from C to Python. To accomplish this task, Futamura projections are used. GraalVM is one of the first production compilers to successfully implement the first Futamura projection. This technique only requires that language implementers produce a functional interpreter. Then, utilizing partial evaluation, the Truffle framework is able to execute the given language interpreter at comparable speeds to a tailor-made compiler.

2 Implementing a Language

To truly understand the advantage afforded by this technique, we demonstrate implementing a new language. The language is very simple, each line represents an arithmetic expression, when the line is evaluated, the resulting value of the expression will be output. An example program in our Simple Arithmetic Language (SAL) is shown in Figure 1.

```
a + b - 0;
a - c;
```

Figure 1: A program in our Simple Arithmetic Language

```
Program program = new Program(
    new SubNode(
        new AddNode(
            new VariableNode("a"),
            new VariableNode("b")
        ),
        new ConstantNode(0)
    ),
    new SubNode(
        new VariableNode("a"),
        new VariableNode("c")
    )
);
```

Figure 2: Constructing an AST to represent the program from Figure 1

To implement SAL, we will assume there is an existing collection of classes for representing the AST. Each expression has a corresponding node class which are nested to form a tree structure.

Figure 2 shows how to construct an AST which represents the program from Figure 1. In addition to an AST, SAL needs an interpreter. The interpreter for SAL uses the visitor pattern to offload execution semantics onto each node class. The interpreter loop for SAL is shown in Figure 3.

Each line, or expression, of the program is executed by calling the `execute` method of the top level node. Figure 4 illustrates a representative collection of `execute` methods for AST nodes. The tree structure allows binary and unary expressions to be easily evaluated using the expected operations. We will treat variables as arbitrary dynamic program data, performing a symbol table lookup is a blackbox which can produce different values during each execution.

```
void executeProgram(Program program,
                    SymbolTable symtab) {
    for (Expression line : program.getExpressions()) {
        int value = line.execute(symtab);
        print(value);
    }
}
```

Figure 3: SAL interpreter main loop

```
class AddNode extends Expression {
    Expression left;
    Expression right;

    int execute(SymbolTable symtab) {
        return left.execute(symtab)
                + right.execute(symtab);
    }
}

class ConstantNode extends Expression {
    int value;

    int execute(SymbolTable symtab) {
        return value;
    }
}

class VariableNode extends Expression {
    String identifier;

    int execute(SymbolTable symtab) {
        return symtab.lookup(identifier);
    }
}
```

Figure 4: Visitor methods of SAL AST node classes

```
Program program = new Program(
    new SubNode(
        new AddNode(
            new VariableNode("a"),
            new VariableNode("b")
        ),
        new ConstantNode(0)
    ),
    new SubNode(
        new VariableNode("a"),
        new VariableNode("c")
    )
);
for (Expression line : program.getExpressions()) {
    int value = line.execute(symtab);
    print(value);
}
```

Figure 5: Visualisation of partially evaluating an interpreter with a source program

3 Partial Evaluation

The interpreter developed in the previous section would yield underwhelming performance if used directly, particularly in comparison to tailor-made compilers. To overcome this inefficiency partial evaluation is used. In partial evaluation, program input is divided into static input data and dynamic input data. Partial evaluation embeds the static input data within the program, this produces a program which only takes dynamic input data. When combined with traditional optimization tactics, this technique can significantly improve performance.

Futamura projections refer to various partial evaluation techniques. Specifically, when the program being partially evaluated (or specialised) is an interpreter. The first futamura projection is implemented by GraalVM. In the first projection the static input data is a language written in the target language of the interpreter. Specialisation of the interpreter will then produce an executable of the input program.

Let us explore how specialisation would be applied to our example SAL program from Figure 1. Figure 5 assists in visualising specialisation. Our static input data, the program AST, is inlined within the original `executeProgram` method from Figure 3.

```
1  int value = new SubNode(
2      new AddNode(
3          new VariableNode("a"),
4          new VariableNode("b")
5      ),
6      new ConstantNode(0)
7  ).execute(symtab);
8  print(value);
9
10 int value = new SubNode(
11     new VariableNode("a"),
12     new VariableNode("c")
13 ).execute(symtab);
14 print(value);
```

Figure 6: Specialised example program after loop unrolling

4 Interpreter Optimization

Partial evaluation by itself does not automatically introduce enhanced efficiency. Partial evaluation is a method for exposing optimizations which can be performed. Now that the interpreter has been specialised so that the value of `program` is known, the interpreter can be optimized for `program`.

First, we know the `program` only has two lines, or expressions. From that an optimizer can perform loop unrolling, see Figure 6. Loop unrolling does not increase efficiency itself, again, it is a step for exposing potential optimizations.

And indeed, it would seem that more optimizations are exposed. From here, it would be pragmatic for an optimizer to begin applying method inlining. One pass of inlining would result in Figure 7. When inlining is applied to the full practical extent, we end up with the program in Figure 8.

If not already clear, Figure 8 should make the power of partial evaluation abundantly clear. Through partial evaluation and optimization, we have produced the Java equivalent of our example program which was originally written in a custom language.

Of course, we (and our optimizer) are not finished. There is one final and obvious optimization to perform. The end of line 3 in Figure 8 has a completely redundant subtraction of zero. We should therefore apply one final optimization to the interpreter. But note; this last optimization has blurred the line between interpreter and source program. We are no longer optimizing the interpreter for the purposes of

```
1  int value = new AddNode(
2          new VariableNode("a"),
3          new VariableNode("b")
4      ).execute(symtab) -
5      new ConstantNode(0).execute(symtab);
6  print(value);
7
8  int value = new VariableNode("a").execute(symtab)
9      - new VariableNode("c").execute(symtab);
10 print(value);
```

Figure 7: Specialised example program after one pass of method inlining

```
1  int value = (
2      symtab.lookup("a")
3      + symtab.lookup("b")) - 0;
4  print(value);
5
6  int value = symtab.lookup("a") - symtab.lookup("c");
7  print(value);
```

Figure 8: Specialised example program after all passes of method inlining

partial evaluation. We have just optimized a redundancy from the original program.

5 Conclusion

We have manually specialised a new language interpreter for an example program. Through this process, we have demonstrated the power and versatility afforded by optimizations in the GraalVM compiler. For hosted GraalVM languages, optimizations perform two important functions;

- enabling efficient interpretation through specialisation, and
- sharing optimizations between languages by further optimizing the specialised interpreter.

Consequently, it is crucial for optimizations to be correct. Any incorrect optimization during the specialisation process could quickly introduce unintelligible program execution. To ensure correct optimizations, we are undertaking a project which aims to formally verification these optimization passes.